VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

3D HRA V ENGINU GODOT

3D GAME IN GODOT ENGINE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MARTIN TRBOLA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. TOMÁŠ MILET,**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Trbola Martin**
Obor: Informační technologie
Téma: **3D hra v enginu Godot**
3D Game in Godot Engine
Kategorie: Počítačová grafika

Pokyny:

1. Prostudujte knihovnu Godot.
2. Nastudujte možnosti procedurálního generování 3D světa.
3. Navrhněte hru s využitím procedurálního generování.
4. Implementujte navrženou aplikaci.
5. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte video pro prezentování projektu.

Literatura:

- Podle pokynů vedoucího

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Milet Tomáš, Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Godot je nový opensource herní engine. Cílem této práce je vytvoření bojové 3D hry v tomto enginu s využitím procedurálního generování pro tvorbu úrovní a nepřátel. Pro implementaci hry byla použita verze Godot 2.1. Procedurální generování levelů je implementováno použitím BSP stromů a generování nepřátel je implementováno spojováním menších částí dohromady.

Abstract

Godot is a new opensource game engine. The goal of this thesis is to create a 3D fighting game in this engine, using procedural generation to create levels and enemies. This game was implemented using Godot version 2.1. The procedural level generation is implemented using BSP trees and the procedural enemy generation is implemented by connecting smaller parts together.

Klíčová slova

hra, 3D, Godot engine, procedurální generování, BSP stromy

Keywords

game, 3D, Godot engine, procedural generation, BSP trees

Citace

TRBOLA, Martin. *3D hra v enginu Godot*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Milet,

3D hra v enginu Godot

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Mileta. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Trbola
14. května 2018

Poděkování

Rád bych poděkoval svému vedoucímu Ing. Tomáši Miletovi za jeho čas, ochotu a dobré rady při vedení této práce.

Obsah

Kapitola 1

Úvod

Herní engine je důležité jádro každé hry, ať už se jedná o veřejně dostupný engine, nebo vývojář napíše engine speciálně pro danou hru. Herní engine abstrahuje prvky, které jsou potřeba v každé hře, nebo ve většině her, jako je například vykreslování, ať už 2D nebo 3D, fyzika, hudba a zvukové efekty, animace, uživatelské rozhraní, uživatelský vstup, skriptování a podobně a zároveň usnadňuje organizaci a práci s těmito prvky.

Nejznámějšími herními enginy v dnešní době mohou být například Unity, Unreal, CryEngine nebo Source Engine.

Godot je poměrně nový herní engine, který ovšem rychle dohání výkon a sílů výše zmíněných herních enginů, hlavně díky svému otevřenému zdrojovému kódu a velkým množstvím aktivních přispěvatelů. V tomto projektu pracuji s trochu starší verzí Godot 2.1.4, ovšem od ledna 2018 je dostupná verze Godot 3.0, která obsahuje velké množství podstatných vylepšení [2].

Cíl této práce je vytvořit 3D hru v tomto herním engine, využívající procedurální generování herního světa.

Tato práce je rozdělena na několik kapitol. Druhá kapitola popisuje engine Godot, obecný přehled o historii a součástech tohoto engine a konkrétní součásti použité v tomto projektu. Třetí kapitola se zabývá návrhem samotné hry, analýzou podobných her a jednotlivými prvky této hry. Ve čtvrté kapitole popisuji konkrétní implementaci navržených systémů a pátá kapitola obsahuje zhodnocení výsledku a možnosti pokračování v tomto projektu.

Kapitola 2

Herní engine Godot

Godot je nový multiplatformní herní engine, který vytvořili Juan Linietsky a Ariel Manzur [3]. Engine je napsaný v jazyce C++ a od roku 2014 je opensource pod MIT licencí [4], která umožňuje volnou modifikaci, šíření i prodej samotného enginu i her v něm vytvořených bez nutnosti placení autorských poplatků [18].

Engine umožňuje tvorbu 2D i 3D her z jednotného uživatelského rozhraní a nabízí velké množství různých funkcí, jako jsou například vykreslování, uživatelské rozhraní a vstup, animace, fyzika, zvuky a další. Hra se dá exportovat pro velké množství platforem a to do hlavních desktopových (Linux, Windows, Mac OS X) a mobilních (Android, iOS) operačních systémů a webu. [10]

V této kapitole popisuji součásti tohoto engine, které jsou podstatné pro tento projekt. Popisuji hlavně verzi Godot 2.1.4, ale občas zmiňuji zajímavé změny dostupné od verze Godot 3 vydané na začátku tohoto roku.

2.1 Programovací jazyky

Godot je napsaný v jazyce C++ a díky otevřenému zdrojovému kódu umožňuje změnit nebo rozšířit funkcionalitu editoru a hry nebo celou hru napsat v tomto jazyce. Jazyku C++ se dá využít buď změnou implementace existujících tříd nebo přidáváním modulů. Nevýhoda tohoto přístupu je ta, že se vždy musí znovu přeložit celý engine, nicméně přidané moduly se dají také přeložit jako sdílený objekt, který se ke hře připojí dynamicky. Godot pro překlad využívá build systém SCons. [5]

Doporučeným způsobem programování v Godot enginu je jazyk GDScript. GDScript je dynamický jazyk hodně podobný jazyku Python speciálně vytvořený pro tento engine. GDScript umožňuje jednoduché rozhraní se samotným enginem a ačkoliv není tak výkonný jako kód napsaný přímo v C++, pro většinu her je naprosto dostačující, přičemž kritické části je možné urychlit pomocí C++. [13]

Každá proměnná v jazyce GDScript je objekt třídy `Variant`, která umožňuje dynamicky ukládat všechny základní datové typy a objekty, porovnávat je navzájem, konvertovat mezi různými typy a podobně [16]. Každý soubor v jazyce GDScript je nepojmenovaná třída, rozšiřující libovolnou třídu enginu nebo jiný soubor v jazyce GDScript, přičemž základní třídou, ze které dědí každý objekt, je třída `Object`. Každý tento skript se dá přiřadit libovolnému objektu, který je stejnou třídou nebo podtřídou třídy, kterou daný skript rozšiřuje. [7]

Třída `Object` a třída `Node`, která je základní třídou všech scén (více popsáno v kapitole 2.2), poskytují několik virtuálních callback funkcí, z nichž nejzajímavější jsou: [14]

- `_init(...)` – konstruktor; volá se při vytvoření objektu, před vytvořením všech potomků, a může mít libovolný počet parametrů,
- `_ready()` – zavoláno jednou po vytvoření objektu a vložením daného objektu do scény, po vytvoření všech potomků,
- `_process(delta)` – zavoláno každý herní cyklus, který se provádí co nejrychleji, takže proměnná `delta` může mít různou hodnotu,
- `_fixed_process(delta)` – zavoláno každý herní cyklus synchronizovaný s fyzikou, takže proměnná `delta` by měla mít stále stejnou hodnotu;
  spolu s funkcí `_process(delta)` se jedná o návrhový vzor *Update Method* [17, Kapitola 10],
- `_input(event)` – zavoláno při uživatelském vstupu, proměnná `event` obsahuje informace o daném vstupu.

Godot verze 3 dále podporuje programování v jazyce C# skrze Mono, vizuální skriptování, urychluje GDScript a umožňuje rozšiřovat engine v jazyce C++ bez nutnosti překládat celý engine a to pro všechny platformy skrze GDNative framework [2].

2.2 Scény

Hry napsané v enginu Godot se skládají z uzlů a scén. Uzly jsou základním stavebním blokem. Každý uzel může mít různou funkcionalitu, má nějaké jméno, editovatelné parametry, může být rozšířený skriptem a může obsahovat další uzly, čímž vzniká stromová struktura. [12]

Skládáním různých typů uzlů dohromady a nastavováním komunikace mezi těmito uzly se vytváří komplexní funkcionalita. Jedná se o návrhový vzor *Component* [17, Kapitola 14].

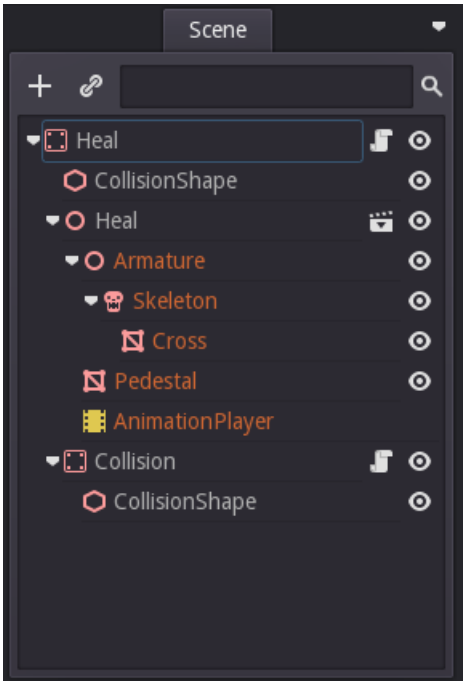
Tyto uzly se skládají do scén, přičemž každá scéna má přesně jeden kořenový uzel, dá se uložit na disk a instanciovat do jiných scén. Každý projekt má nastavenou hlavní scénu a spuštění hry znamená spuštění této scény. [12]

Existují různé typy uzlů, jako jsou například 2D objekty a GUI, 3D objekty, uzly pro přehrávání animací a zvuků a podobně. Ukázka vytvořené scény je na obrázku 2.1.

2.3 Signály

Signály jsou důležitým způsobem komunikace mezi různými uzly. Jedná se o verzi návrhového vzoru *Observer* [17, Kapitola 4], kdy každý uzel může být zároveň *observer* i *observable*. Toto řešení umožňuje uzlům mezi sebou komunikovat, aniž by o sobě vzájemně věděly. Každý signál může mít definovány argumenty, které se mohou odeslat buď při poslání daného signálu nebo se mohou nastavit přímo při spojování dvou uzlů.

Spousta tříd enginu Godot definuje některé signály, ale tato funkcionalita se dá rozšířit i vlastními signály.



Obrázek 2.1: Ukázka uzlů a scén; uzel „Heal/Heal“ je instance jiné scény s editovatelnými potomky

2.4 Vstup

Godot nabízí abstrakci nad uživatelským vstupem, která nevyžaduje přímou komunikaci s operačním systémem. Pokud aplikace obdrží vstup od uživatele, tento vstup je odeslán všem uzlům, které vstup zpracovávají, pomocí objektu třídy `InputEvent`. Tento objekt má uloženou informaci o typu vstupu, zda se jedná o klávesnici, myš, joystick nebo, na mobilních zařízeních, dotek a podle daného typu obsahuje další informace o tomto vstupu, jako je například konkrétní klávesa, tlačítko myši a podobně. [8]

Godot dále nabízí další abstrakci nad uživatelskými vstupy v podobě akcí. Každý projekt může mít nastaveno několik různých akcí a každá akce může mít přiřazeny různé typy vstupu, jako je klávesnice, myš nebo joystick, což umožňuje jednotný kód, nezávisle na konkrétním vstupním zařízení. Toto nastavení se dá dynamicky měnit při běhu programu. [8]

2.5 Fyzika

Godot pro fyziku a kolize používá vlastní fyzikální engine, který je přímo integrovaný do systému scén.

Existují čtyři základní druhy fyzikálních objektů:

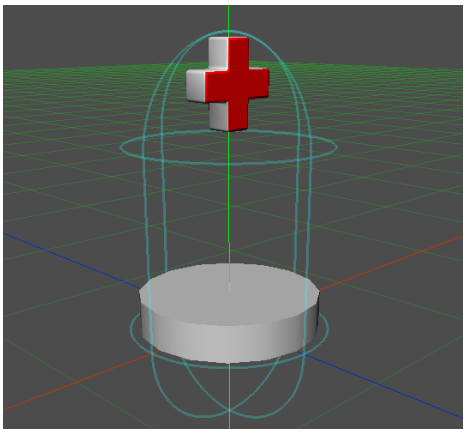
- *StaticBody* – fyzikální těleso, které je zapojeno do kolize mezi objekty, ale po kolizi se nepohybuje a není ovlivněno fyzikou; používá se převážně pro zdi herní úrovně.
- *KinematicBody* – toto fyzikální těleso je součástí kolizí mezi objekty a není ovlivněno fyzikou, ale může se pohybovat pomocí kódu, přičemž se mu při pohybu počítá odhad lineární a úhlové rychlosti; lze použít například pro pohybující se plošiny.

- *RigidBody* – typ fyzikálního tělesa, které je ovlivňováno fyzikou; má definovanou váhu, tření a pružnost, přičemž počátek souřadného systému ovlivňuje střed hmotnosti; používá se pro všechny objekty, které potřebují fyziku.
- *Area* – definuje oblast, která není ovlivňována fyzikou a neúčastní se kolizí mezi objekty, ale může detekovat všechny výše zmíněné tělesa vstupující nebo opouštějící danou oblast.

Aby každý z výše uvedených fyzikálních objektů mohl interagovat s ostatními objekty, musí mít přiřazen alespoň jeden kolizní tvar. Godot nabízí několik základních tvarů, jako jsou například kvádr nebo koule ve 3D a obdélník nebo kruh ve 2D. Ukázka 3D kolizního tvaru je na obrázku 2.2.

Dále existuje objekt *RayCast*, který umožňuje detekovat ostatní fyzikální tělesa vysláním paprsku v určitém směru. [9]

Godot 3 pro fyziku využívá opensource fyzikální engine Bullet [2].



Obrázek 2.2: Ukázka kolizního tvarů pro kolizi okolo interaktivního objektu

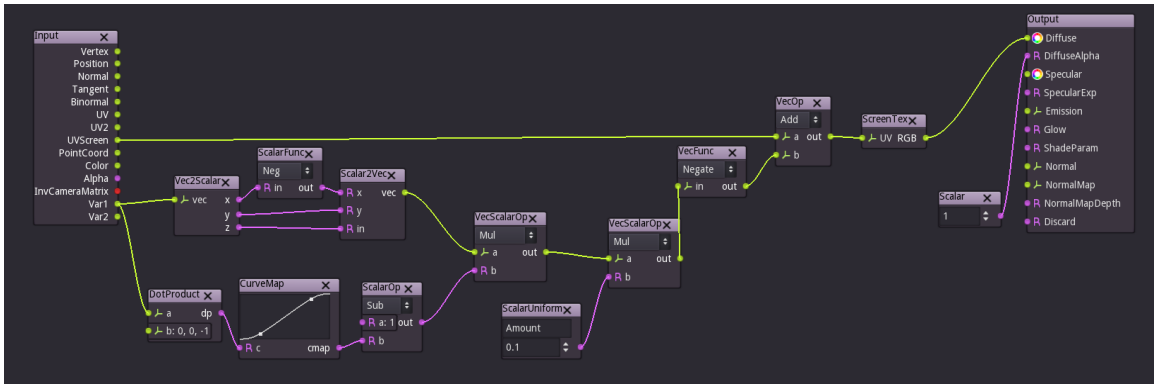
2.6 Vykreslování

Vykreslování 3D scén v Godot 2.1 je velmi základní a poměrně omezené ve srovnání s ostatními moderními herními enginy. Každý mesh, nebo části meshe, mohou mít nějaký materiál, přičemž existují dva základní druhy materiálů: *Fixed material* a *Shader material* [11].

Fixed material je základní typ materiálu, nabízející nejčastější možnosti dostupné ve 3D editorech, jako jsou *diffuse*, pro základní barvu objektu, *specular* pro odlesky nebo *emission* a *glow* pro svit objektu. Tyto parametry se dají nastavit buď jako jedna barva nebo jako textury. Tento materiál dále nabízí čtyři shadery a to *Lambert*, *Wrap*, *Velvet* a *Toon* a také lze na tento materiál použít normálovou mapu [6].

Shader material využívá jednoduchého shaderovacího jazyka, který je téměř podmnožinou GLSL. Lze použít pro procedurálně generované textury, animované materiály, animované vrcholy modelu a další pokročilé efekty. Protože je tento jazyk jednoduchý, Godot nabízí i možnost tvorby těchto materiálů pomocí grafového editoru [15]. Ukázka takového shaderu je na obrázku 2.3.

Vykreslování v Godot 3 bylo úplně změněno a nahrazeno nejmodernějším *physically based* vykreslováním, které nabízí i některé funkce, které zatím nejsou běžné v ostatních herních enginech a dále nabízí i spoustu *mid-processing* a *post-processing* možností.



Obrázek 2.3: Ukázka shader grafu pro efekt tlakové vlny

2.7 Animace

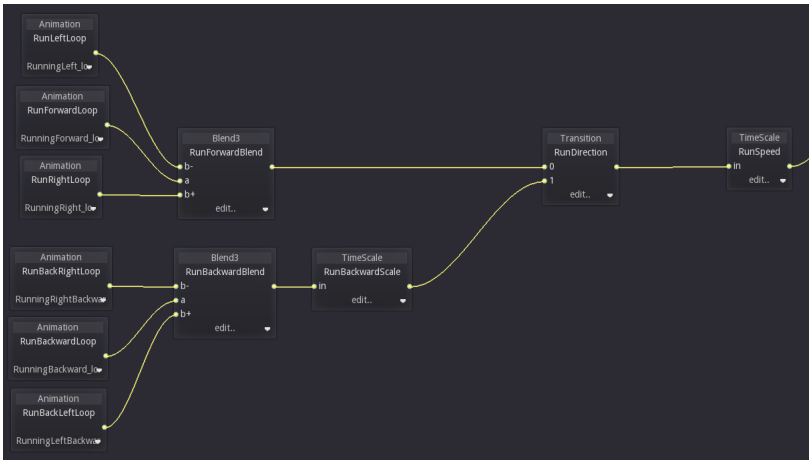
Animování v Godot enginu je řešeno pomocí třídy `AnimationPlayer`. Tato třída může obsahovat libovolný počet stop, kde každá stopa má nastavený cíl modifikace a obsahuje libovolný počet klíčových snímků. Existují tři typy stop a to *Transform*, *Parameter* a *Method*.

Transform stopy upravují transformaci libovolného objektu ve scéně. Tyto stopy se dají používat i pro skeletální animace, kdy cílem této stopy je konkrétní kost.

Parameter stopy mohou upravovat parametry libovolných objektů, přičemž tyto změny mohou být prováděny diskrétně nebo spojitě s několika možnostmi interpolace.

Method stopy mohou v definovaných klíčových snímcích volat funkce libovolného objektu.

Godot také nabízí animování pomocí stromu animací. Tyto stromy nejprve potřebují objekt třídy `AnimationPlayer`, ve kterém jsou uložené jednotlivé animace, a tyto animace se následně dají přehrávat a směšovat pomocí propojovaných uzlů. Ukázka takového animačního stromu je na obrázku 2.4.



Obrázek 2.4: Ukázka animačního stromu pro animace běhu postavy

Kapitola 3

Návrh hry

Hra je navržená jako bojová hra s procedurálním generováním úrovní a nepřátel. V této kapitole analyzuji hry využívající těchto herních mechanik a podobné hry a popisuji návrh finální podoby tohoto projektu.

3.1 Podobné hry

3.1.1 Binding of Isaac

Tato hra využívá procedurálního generování úrovní a obsahuje spoustu nepřátel, přičemž každá úroveň je zakončená bossem. Hra dále obsahuje velké množství věcí, které herní postavu vylepšují a graficky upravují. Postava se pohybuje klávesami WSAD a střílí ve směru kurzorových šipek. Ukázka z této hry je na obrázku 3.1.

Obrázek 3.1: Ukázka ze hry Binding of Isaac

3.1.2 Robot Legions

Jedná se o malou bojovou hru. Hra se odehrává v plochém obdélníkovém světě a obsahuje množství nepřátel, z nichž každý má unikátní styl boje. Za každého zničeného nepřítele hráč dostává body, za které může vylepšovat herní postavu. Postava se pohybuje klávesami WSAD a střílí ve směru kurzoru myši. Ukázka z této hry je na obrázku 3.2.

Obrázek 3.2: Ukázka ze hry Robot Legions

3.1.3 Starbound

Přežívací sandbox hra s procedurálně generovanými nepřáteli. Nepřátelé se dělí do několika různých tříd, podle jejich velikosti, způsobu pohybu a agrese. Mohou mít různý počet životů a způsobovaného zranění a také různé způsoby boje. Ukázka z této hry je na obrázku 3.3.

Obrázek 3.3: Ukázka ze hry Starbound

3.2 Prvky hry

Hra se bude odehrávat v procedurálně generovaném uzavřeném vnitřním prostředí, podobném patrům budov. Hlavní postavou je kyborg, který bojuje proti procedurálně generovaným robotům.

Vygenerovaní nepřátelé budou spadat do několika různých tříd, kde každá třída bude mít jiný způsob boje. Na konci každé úrovně bude boss, kterého bude nutno zabít pro postup do další úrovně.

Hlavní postava bude vybavena mečem a lukem a bude mít k dispozici několik speciálních útoků, přičemž použití těchto útoků a luku bude stát energii, která se bude generovat soubojem pomocí meče. Po zabití nepřítele hráč dostane body, za které bude moct postavu vylepšovat pomocí stromu vylepšení.

V úrovních budou kromě nepřátel i interaktivní objekty nabízející různé účinky, například vyléčení hráče, bonus bodů pro vylepšení, postup do další úrovně a podobně. Budou zde i interaktivní objekty obsahující datové karty, kterou hráč může mít jednu v inventáři a po použití dostane nějaký dočasný efekt, jako například vyšší rychlost, regeneraci a podobně. Některé interaktivní objekty budou sloužit pro vylepšování zbraní.

Postava se bude ovládat klávesami WSAD nebo kurzorovými šipkami a bude se otáčet ve směru kurzoru myši.

Kapitola 4

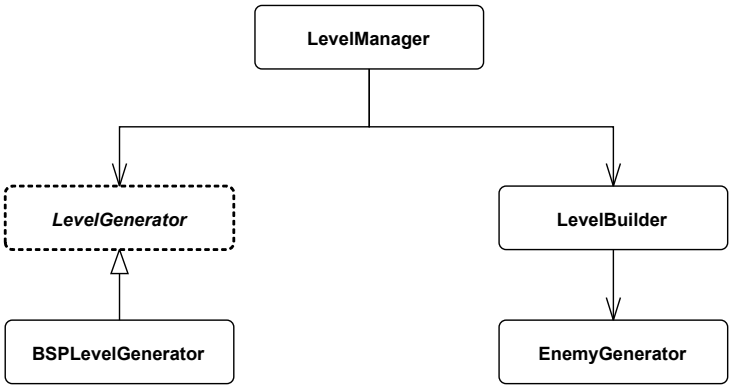
Implementace hry

Ačkoliv bylo potřeba opravit a upravit několik věcí přímo v enginu, celá hra je napsaná v jazyce GDScript. Implementace celé hry se dá rozdělit do jednotlivých, vzájemně komunikujících, částí. V této kapitole popisuji implementaci jednotlivých částí, i hry jako celku.

4.1 Procedurální generování úrovní

Procedurální generování úrovní je rozděleno na dvě třídy. Třída `LevelGenerator` generuje úroveň a vrací strukturu obsahující abstraktní informace o rozměrech úrovně, jednotlivých místnostech, počáteční pozici hráče a umístění nepřátel a objektů. Třída `LevelBuilder` následně tyto informace použije pro vytvoření meshů a kolizí úrovně, vytvoření navigačního meshe pro nepřátele a instanciaci a umístění konkrétních nepřátel a objektů. O komunikaci mezi těmito třídami se stará třída `LevelManager`. Třída `LevelBuilder` zároveň využívá třídy `EnemyGenerator` pro generování nepřátel.

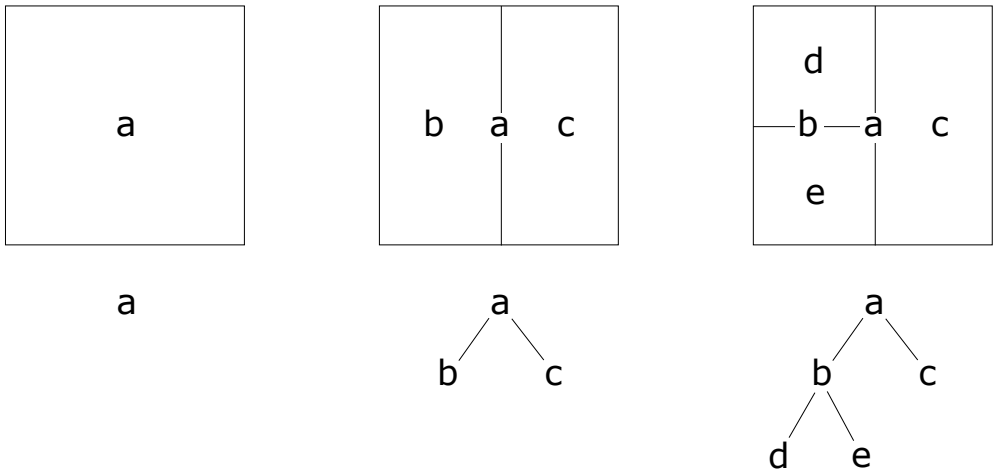
Tento návrh umožňuje jednoduchou změnu generovacího algoritmu bez nutnosti změny algoritmu pro převod úrovně do 3D. Na obrázku 4.1 je diagram těchto tříd a jejich vzájemných vztahů. Samotné generování úrovně je vyřešeno pomocí BSP stromů.



Obrázek 4.1: Diagram tříd pro generování úrovní

4.1.1 Generování pomocí BSP stromů

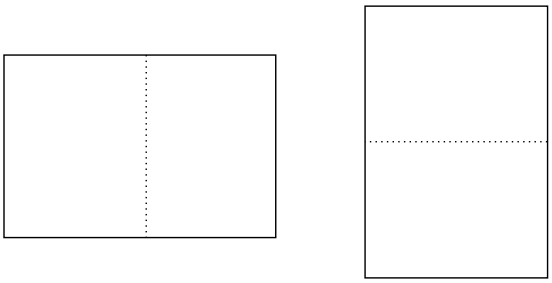
BSP stromy (neboli *Binary Space Partitioning* stromy) jsou metoda rekurzivního dělení n-rozměrného prostoru nadrovinou na dvě části. Ukončovací podmínka tohoto procesu, i konkrétní nadroviny použité pro dělení, závisí na konkrétní aplikaci a mohou být libovolné. Výsledkem tohoto procesu je binární strom [20]. Na obrázku 4.2 je ukázka tvorby dvojrozměrného BSP stromu pomocí ortogonálních přímek.



(a) Krok 1 (b) Krok 2 (c) Krok 3

Obrázek 4.2: Ukázka tvorby BSP stromu

Při generování úrovní má každá úroveň stejnou velikost a celá tato plocha je použita jako kořenový uzel vytvářeného BSP stromu. Každý uzel se rozděluje pouze ortogonálními přímkami, jejichž orientace závisí na rozměrech daného uzlu; pokud má uzel větší šířku, než výšku, rozděluje se vertikálně, jinak horizontálně (viz obrázek 4.3).



(a) Vertikálně (b) Horinzontálně

Obrázek 4.3: Ilustrace preference rozdělení uzlu

Pozice rozdělovací přímky se vybírá náhodně, s normálním rozložením pravděpodobnosti, přičemž vzdálenost této přímky od obou rovnoběžných stran je minimálně 20 % ortogonálních stran, nebo definovaná minimální vzdálenost.

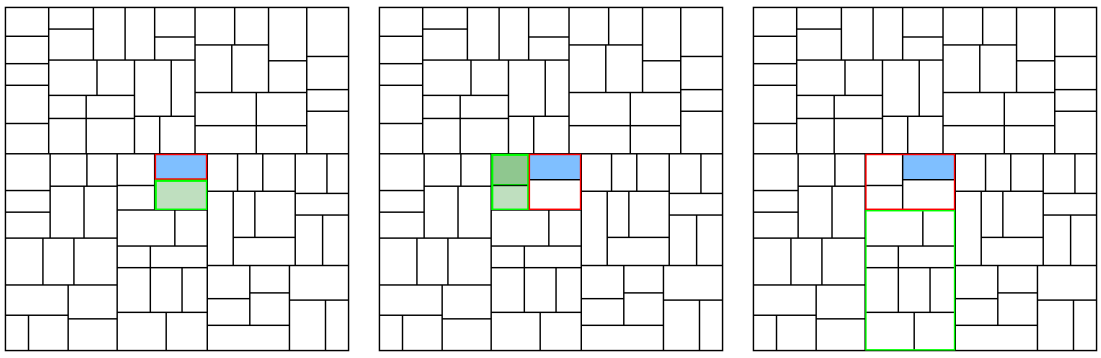
Uzel se nerozděluje, pokud je jeho obsah menší, než je definovaný minimální obsah uzlů. Ukázka takto rozdělené plochy je na obrázku 4.5a.

Při rozdělení uzlu se vypočítají pozice potomků v rozdělovaném uzlu (vpravo, vlevo, nahoře, dole).

Po rozdělení plochy se vypočítá graf všech sousedních uzlů, nejen listů. Sousední uzly se počítají rekurzivně od kořene stromu pro každý uzel zvlášť.

Od každého uzlu, kterému počítám sousední uzly, se postupně vynořuji směrem ke kořenovému uzlu a testuji, jestli pozice sousedního uzlu v BSP stromu je taková, na které zatím nemám vypočítané žádné sousedy. Pokud ano, vím, že daný uzel se musí dotýkat uzlu, kterému počítám sousedy. V takovém případě tento uzel přidám do seznamu sousedů a postupně se rekurzivně zanořuji dovnitř, přičemž testuji, zda se uzly dotýkají s počítaným uzlem, pomocí jejich pozice a souřadnic, a to tak, aby se uzly dotýkaly alespoň minimální definovanou délkou. Tento proces je krok po kroku ilustrován na obrázku 4.4.

Vypočítaný graf pouze sousedních listových uzlů je na obrázku 4.5b.



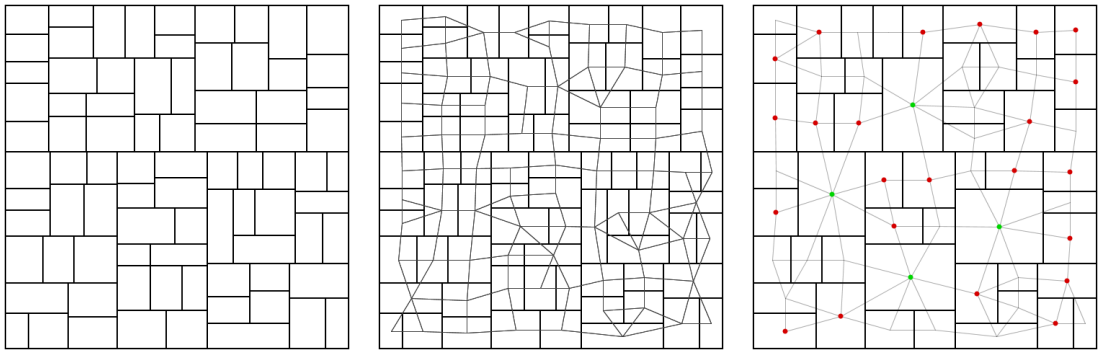
(a) Krok 1 (b) Krok 2 (c) Krok 3

(d) Krok 4 (e) Krok 5 (f) Krok 6

Obrázek 4.4: Ilustrace výpočtu sousedních uzlů. Uzel, kterému se počítají sousední uzly, je zvýrazněný modře. Červeně orámovaný je aktuální uzel v daném kroku a zeleně orámovaný je jeho sousední uzel v BSP stromu. Zeleně zvýrazněny jsou nalezené sousední uzly v daném kroku.

Po vypočítání všech sousedních uzlů jsou vytvořeny dva typy speciálních místností, lišící se velikostí a funkcí (na obrázku 4.5c zvýrazněny červeným a zeleným bodem), odstraňováním podstromů a to tak, aby zbylý listový uzel nepřekročil definovaný poměr stran a byl nejmenším uzlem v daném podstromu, který má větší obsah, než je minimální definovaný obsah pro daný typ speciální místnosti.

Předtím vypočítaný graf sousedních uzlů je použit k tomu, aby větší speciální místnosti (na obrázku 4.5c zvýrazněny zeleným bodem) nebyly sousedé.



(a) Rozdělení prostoru (b) Sousední listy (c) Speciální místnosti

Obrázek 4.5: Rozdělení prostoru, výpočet sousedních uzlů a vytvoření speciálních místností

Zbylé listové uzly budou použity jako chodby mezi těmito místnostmi.

Po vytvoření speciálních místností se vytváří cesta mezi všemi listovými uzly. Nejprve se náhodně vybere počáteční listový uzel, ze kterého se bude cesta generovat, přičemž nejvyšší prioritu mají uzly, ze kterých budou chodby, následně menší typ speciálních místností (na obrázku 4.5c zvýrazněny červeným bodem) a poté vše ostatní, a tento uzel se vloží do seznamu aktivních uzlů.

V každém cyklu generování cesty se ze seznamu aktivních uzlů náhodně, se stejnými prioritami jako u prvního uzlu, vybere jeden ze sousedních listových uzlů a do dalšího seznamu se uloží všechny sousední listové uzly tohoto uzlu, které zatím nejsou spojeny žádnou cestou. Z tohoto druhého seznamu se opět náhodně, se stejnými prioritami, vybere jeden ze sousedních uzlů, který se vloží do seznamu aktivních uzlů, a tyto dva uzly se vzájemně spojí. Tento cyklus pokračuje, dokud seznam aktivních uzlů není prázdný.

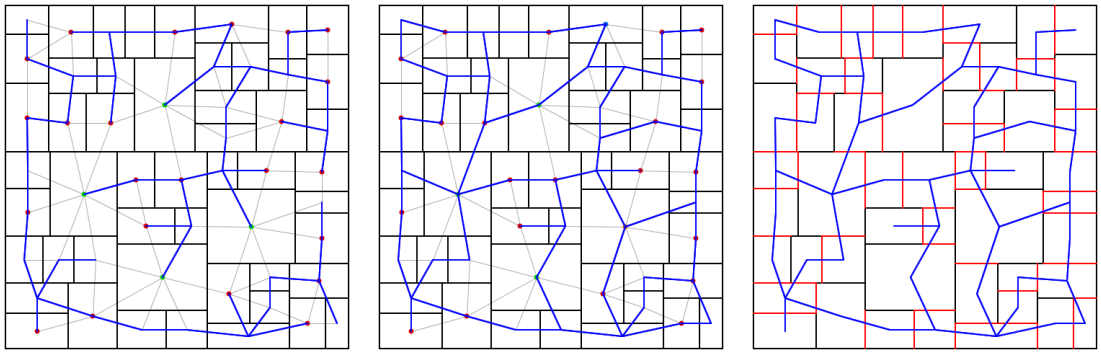
Protože každý uzel spojujeme se sousedním uzlem, který zatím není součástí cesty, ve výsledné cestě nejsou žádné cykly a díky prioritám při náhodném vybírání uzlů vznikají delší chodby, se speciálními místnostmi častěji na koncích cest, než uprostřed, což vytváří zajímavější prostředí. Příklad vygenerované cesty je na obrázku 4.6a.

Po vytvoření cesty se náhodně vybere libovolná menší speciální místnost, která se označí jako počáteční pozice hráče a jako cíl se vybere větší speciální místnost, která má pouze jeden vstup a nejdelší kratší stranu.

Následně se cesta začne spojovat do smyček tak, aby nejdelší vzdálenost mezi libovolnými dvěma uzly byla menší nebo rovna maximální definované vzdálenosti a aby každá chodba měla alespoň dva vstupy.

Pro každý listový uzel se metodou BFS (*Breadth-First Search* – prohledávání do šířky) vypočítá vzdálenost ke všem sousedním listovým uzlům, které s daným uzlem nejsou přímo spojeny a hledá se největší vzdálenost mezi libovolnými dvěma uzly. Tyto dva uzly se spojí a tento cyklus se opakuje, dokud se dají vytvářet nové smyčky. Pokud v cestě zbyly některé chodby, které mají pouze jeden vstup, jsou tyto místnosti označeny jako prázdné. Ukázka vygenerované cesty se smyčkami je na obrázku 4.6b.

Následně se vypočítají sdílené stěny mezi propojenými uzly, na obrázku 4.6c zvýrazněné červeně. Tyto informace jsou použity nejprve k vyrovnání propojovacích chodeb mezi místnostmi a později k vytvoření samotných propojovacích chodeb.



(a) Vygenerovaná cesta (b) Cesta se smyčkami (c) Sdílené stěny

Obrázek 4.6: Generování cesty a výpočet sdílených stěn

Nejjednodušší způsob vytváření propojovacích chodeb je vytvořit je uprostřed každé sdílené stěny. To ovšem vede k vytváření velice křivých propojovacích chodeb, jak lze vidět na obrázku 4.8a.

Pro vyrovnání těchto chodeb nepracuji s chodbami samotnými, ale zmenšuji sdílené stěny tak, aby protější stěny měly stejnou délku a pozici a tím pádem i stejný střed. Stěny se vždy zmenšují na takovou délku a pozici, kterou obě strany sdílí.

Toto zmenšování se opakuje, dokud existují páry protějších stěn, které lze upravit, přičemž páry stěn musí sdílet alespoň minimální definovanou délku. Páry stěn se zmenšují podle následujících priorit:

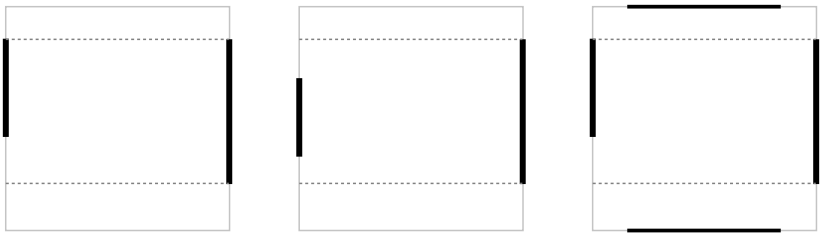
1. Páry stěn, které začínají nebo končí na stejné pozici, v uzlech, kde nejsou žádné ortogonální stěny,
2. páry stěn, kdy délka a pozice jedné stěny je úplně obsažená v druhé stěně, v uzlech, kde nejsou žádné ortogonální stěny,
3. ostatní páry stěn, které začínají nebo končí na stejné pozici,
4. ostatní páry stěn, kdy délka a pozice jedné stěny je úplně obsažená v druhé stěně,
5. ostatní páry stěn, v uzlech, kde nejsou žádné ortogonální stěny,
6. ostatní páry stěn.

Tyto priority jsou ilustrované na obrázku 4.7.

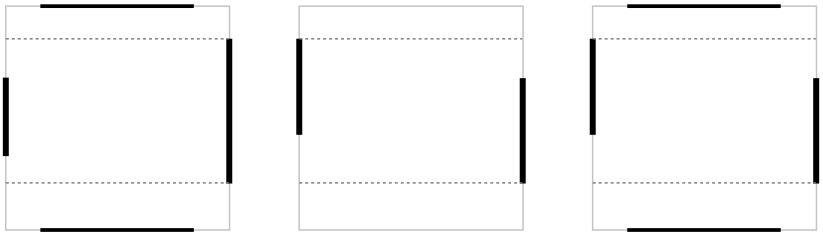
Speciálním případem při vyrovnávání stěn je, pokud na protějších stranách jsou různé počty stěn. V takovém případě se stěny vyrovnávají pouze pokud lze jednoznačně určit konkrétní páry stěn. Pokud lze jedné stěně přiřadit více než jeden pár, tato stěna se nevyrovnává.

Ukázka vyrovnaných stěn je na obrázku 4.8b. Na tomto obrázku lze vidět i speciální případ nevyrovnaných stěn v levém spodním rohu.

Úplně nakonec se vytvoří konkrétní místnosti v každém uzlu a spojovací chodby mezi nimi. Pozice a velikosti speciálních místností se tvoří převážně náhodně, přičemž jsou omezeny definovanou minimální a maximální vzdáleností od okrajů uzlu a také pozicí spojovací chodby. Místnost definovaná jako cíl vyplňuje celý uzel, s definovanou vzdáleností od všech okrajů.



(a) 1 - Nejvyšší priorita (b) 2 (c) 3

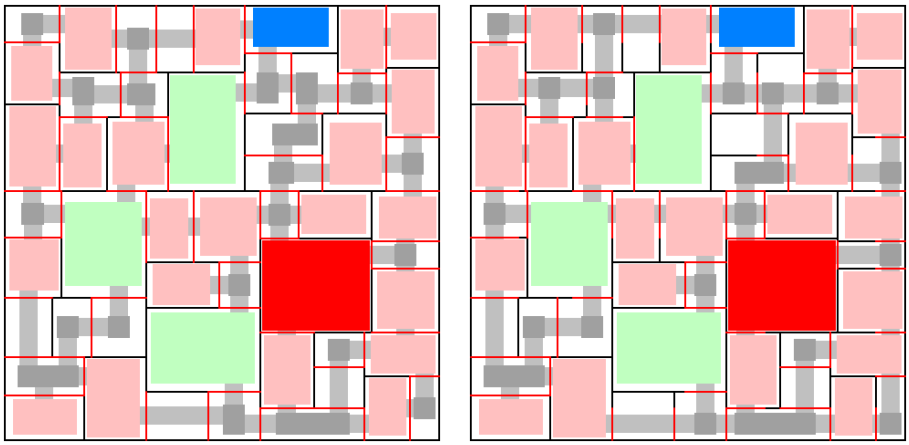


(d) 4 (e) 5 (f) 6 - Nejnižší priorita

Obrázek 4.7: Ilustrace priorit při vyrovnávání stěn

Uzlové místnosti spojující několik spojovacích chodeb se tvoří podle jejich pozicí. Tyto místnosti se tvoří pouze v případě, že jsou potřeba vytvořit. Na obrázku 4.8b lze vidět několik uzlů bez těchto místností.

Spojovací chodby mezi místnostmi se tvoří uprostřed sdílených stěn a mají vždy stejnou definovanou šířku.



(a) Nevyrovnané chodby (b) Vyrovnané chodby

Obrázek 4.8: Vyrovnání chodeb

4.1.2 Rozmístění herních objektů

Do vygenerované úrovně je následně potřeba rozmístit herní objekty. Herní objekty, které se rozmisťují, jsou nepřátelé a interaktivní objekty.

Ve hře jsou momentálně dva interaktivní objekty, ačkoliv hru lze snadno rozšířit o další. Objekty ve hře jsou objekt, který vyléčí hráče a objekt pro postup do další úrovně.

Objekt pro postup do další úrovně je umístěn doprostřed cílové místnosti. Do středů větších speciálních místností se umístí náhodně vybrané interaktivní objekty, kromě objektu pro postup do další úrovně (momentálně pouze vyléčení).

Nepřátelé se umisťují na náhodné pozice do všech speciálních místností. Počet nepřátel v každé místnosti je vypočítán podle definované hustoty nepřátel, přičemž nepřátelé se umisťují s dostatečnou vzdáleností od sebe a od případných interaktivních objektů. Pokud je vzdálenost vybrané pozice nepřítele od počáteční pozice hráče menší, než je definovaná minimální vzdálenost, nepřítel se na danou pozici neumístí.

Každý nepřítel má svoji celočíselnou úroveň, podle které se počítá obtížnost daného nepřítele. Úroveň nepřítele se vybírá náhodně s uniformním rozložením pravděpodobnosti, přičemž minimum je tři úrovně pod aktuální úrovní a maximum se vypočítá podle vzdálenosti nepřítele od počáteční pozice hráče, až do šesti úrovní nad minimem. Pokud je úroveň nepřítele menší než nula, je tato úroveň nastavena na nulu.

4.1.3 Převod do 3D

Po vygenerování místností a umístění objektů a nepřátel je potřeba tyto metainformace převést do 3D. Protože každá místnost má uložené informace pouze o svých rozměrech a sousedních místnostech, je nejprve potřeba vypočítat zdi tak, aby se dalo pohybovat mezi těmito sousedními místnostmi.

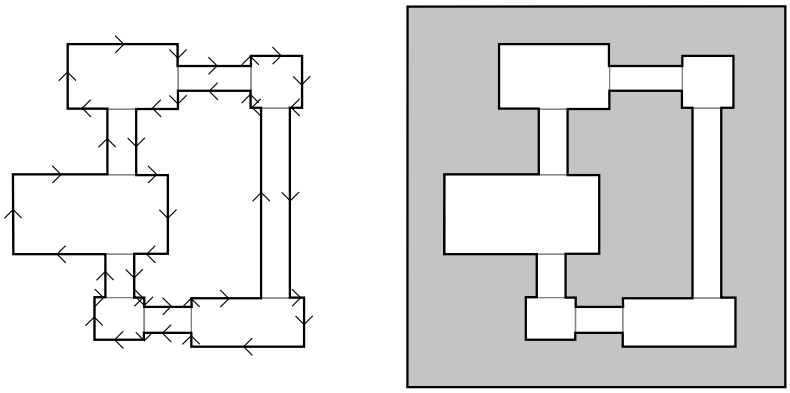
Pro každou místnost se nejprve uloží informace o všech čtyřech zdech, které tuto místnost ohraničují. Následně se u každé místnosti iteruje přes všechny sousedy a pokud je stěna místnosti delší, než dotýkající se stěna souseda, je tato stěna rozdělena a sousedovi je stěna odstraněna. Tento proces by nefungoval v případě, že se místnosti dotýkají pouze částmi stěn, avšak tento případ při výše popsaném generování pomocí BSP stromů nikdy nenastane a není potřeba jej řešit. Vypočítané zdi jsou poté převedeny na indexy do pole všech vrcholů. Všechny zdi jsou uložené po směru hodinových ručiček vzhledem k příslušné místnosti. Toto je ilustrováno na obrázku 4.9a.

Následně se vypočítají polygony výplní mezi zdmi. Vytvoří se objekt, kde se pro každý index vrcholu uloží index následujícího vrcholu. Protože všechny zdi jsou uložené v jednotném směru, pokud začneme na libovolném indexu a budeme přecházet na následující indexy, v konečném počtu kroků se dostaneme zpátky na stejný index. Tohoto faktu se využívá pro výpočet každého polygonu. V každém tomto kroku se daná zeď zároveň odstraňuje z tohoto objektu a tento proces se opakuje, dokud tento objekt není prázdný.

Zároveň s vytvářením těchto polygonů se hledá polygon obsahující vrchol nejvíce vlevo nahoře. Tento polygon je poté rozšířen o několik dalších vrcholů, vytvářejících široký okraj kolem celé úrovně. Tyto polygony jsou ilustrovány na obrázku 4.9b.

Nakonec před samotným generováním 3D meshů a vytvářením instancí objektů se vygenerují šablony nepřátel. Tyto šablony se vytváří pro každou úroveň nepřítele, pokud pro danou úroveň ještě šablona není vytvořená. Samotné generování nepřátel je popsáno v kapitole 4.2.

Pro samotné generování 3D se využívá pomocné třídy Godotu `SurfaceTool`, která umožňuje jednoduše vytvářet meshe a nastavovat jim normálové vektory a materiály.

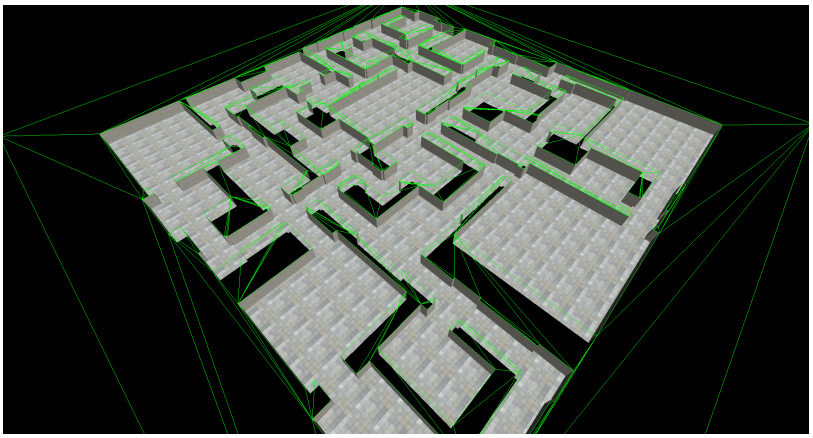


(a) Vypočítané zdi (b) Vypočítané polygony

Obrázek 4.9: Vypočítání zdí a polygonů

Nejprve se vytvoří místnosti a jejich kolize. K vytvoření místností se využívá jejich rozměrů a předtím vypočítaných zdí. Země se vytváří pro každou místnost zvlášť jako jeden mesh a všechny zdi každé místnosti jako druhý mesh. Nevytváří se jeden velký mesh pro celou úroveň. To později může být použito pro optimalizaci vykreslování pomocí místností a portálů.

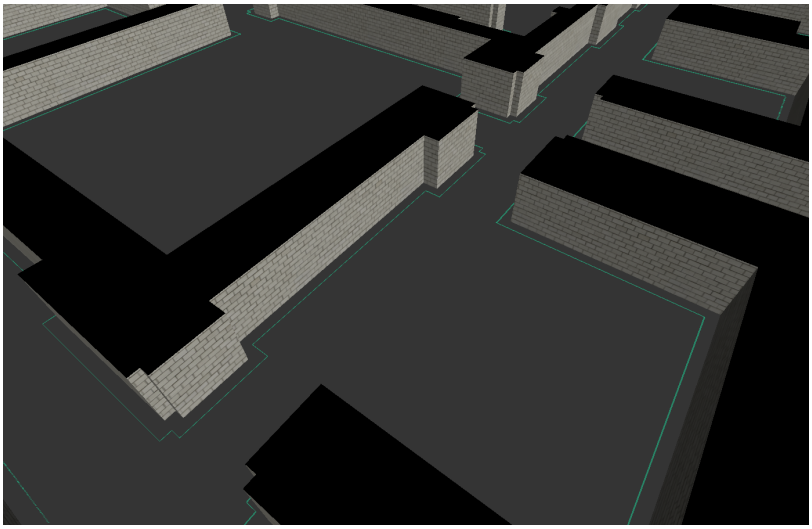
Po vytvoření meshů místností se vytvoří meshe výplní mezi nimi. K tomu se používají předtím vypočítané polygony. Protože třída `SurfaceTool` neumí automaticky převést libovolný obecný polygon na trojúhelníky, jsou tyto polygony triangulovány metodou *ear clipping* [1]. Ukázka vytvořených místností a triangulovaných výplní je na obrázku 4.10.



Obrázek 4.10: Ukázka úrovně ve 3D

Před přidáním nepřátel se vytvoří navigační mesh, který budou všichni nepřátelé používat pro navigaci v úrovni. Navigační mesh se tvoří spojováním polygonů jednotlivých místností. Protože nepřátelé mají určitý poloměr kolizí, pokud by se navigační mesh přímo dotýkal všech zdí, nepřátelé by se mohli zasekávat o rohy. Proto se každý vrchol posune o definovanou vzdálenost ve směru normály obou zdí, které se daného vrcholu dotýkají. Ukázka vytvořeného navigačního meshe je na obrázku 4.11.

Nakonec jsou přidány instance všech objektů a nepřátel na příslušné pozice a tím generování úrovně končí.



Obrázek 4.11: Ukázka navigačního meshe

4.2 Procedurální generování nepřátel

Generování nepřátel je vyřešeno skládáním jednotlivých částí do jednoho celku. Jsou definovány čtyři druhy částí a to: tělo, ruce, nohy, hlava. Každý z těchto typů částí poté může mít svůj podtyp.

Každá část má svůj popis v generovacím skriptu, který uvádí nejen příslušnou scénu s daným modelem a animacemi, ale i další potřebné informace. U všech částí může být uveden limit obtížnosti nepřítele, od které se daná část dá použít, a pravděpodobnost, že se daná část vybere.

U těla se uvádí povolené podtypy končetin, jsou zde popsány jednotlivé konektory, kam se tyto končetiny připojují, včetně informace na které straně daný konektor je a offsetu animací. Dále jsou zde uvedeny maximální počet životů, které toto tělo má, a třída, kterou výsledný nepřítel bude mít.

U nohou se uvádí jejich typ, rychlost běhu, jejich výška pro jednotlivé animace a výchozí výška, pokud není pro některou animaci výška definována.

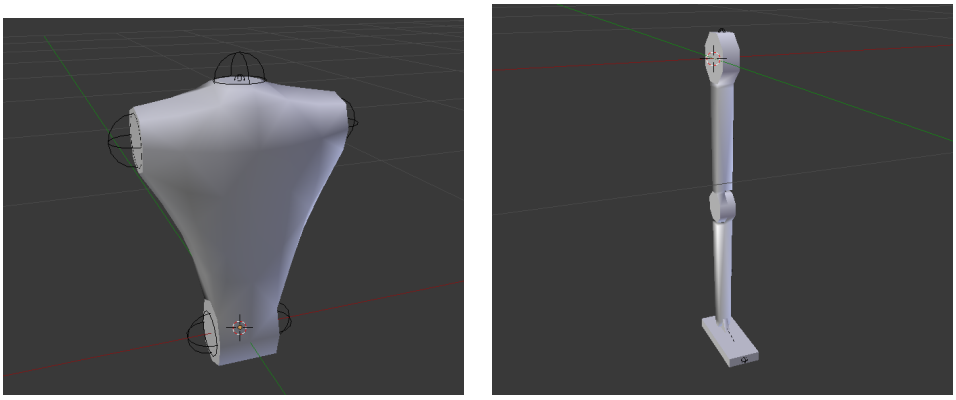
U rukou se uvádí jejich typ, způsobené poškození, dosah a čísla snímků tvořící rozsah, kdy útok zraňuje.

Základem každého generovaného nepřítele je tělo. Model by měl obsahovat prázdné objekty na místech, kam se později připojují končetiny a všechny tyto objekty by měly být popsané.

Na obrázku 4.12 jsou ukázané příklady některých částí.

Protože je každá část animovaná zvlášť, jsou definovány jednotné názvy animací pro každou akci. Ačkoliv Godot umí animovat konkrétní objekty, kvůli technickým nedostatkům Blenderu, ve kterém byly tyto části vytvářeny, jsou všechny animace tvořeny pomocí kostí. To ovšem také znamená, že se tělo nedá animovat, protože prázdné objekty, použité pro spojování částí, se nedají ovládat kostmi. Tento problém by se ovšem dal vyřešit nahrazením prázdných objektů za další kosti, na které by se končetiny připojily.

Každá část má svůj vlastní přehrávač animací, což je poměrně nepraktické, protože při přehrávání jakékoliv animace by se muselo přistupovat ke všem těmhle přehrávačům.



(a) Ukázka těla (b) Ukázka nohy

Obrázek 4.12: Ukázka částí pro generování nepřátel

Některé části ani nepotřebují všechny animace, což by mohlo vést k chybám při pokusech o spuštění neexistujících animací.

Řešením je spojení všech animací do jednoho přehrávače. Toto řešení má tu výhodu, že přímo během spojování se mohou sjednotit délky animací jednotlivých částí a posunout animace podle definicí konektorů. Animace všech částí jsou tak lépe synchronizovány. Zároveň se mohou přidat do animací stopy ovládající výšku modelu a útok nepřítele.

Po sestavení částí a spojení animací do jednoho modelu se duplikují všechny meshe a jejich materiály, kterým se nastaví náhodná barva. Meshe a materiály je nutné duplikovat, jinak by všichni vygenerovaní nepřátelé, používající stejné části, přebarvili všechny ostatní nepřátele, kteří tyto části používají.

Výsledný model, spolu s informacemi o maximálním počtu životů, rychlosti a podobně, získaných z popisů jednotlivých částí, se uloží do třídy `EnemyTemplate`, odkud se při stavění úrovně vytvářejí nové instance těchto nepřátel. Protože každý nepřítel stejného typu využívá stejného meshe pro vykreslování, jedná se o návrhový vzor *Flyweight* [17, Kapitola 3].

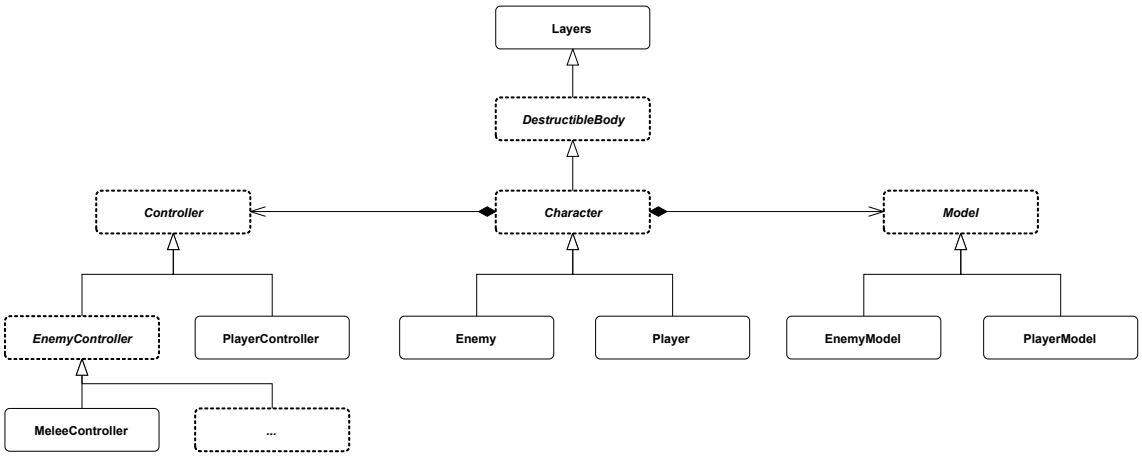
4.3 Vrstvy

Protože engine Godot podporuje různé vrstvy pro fyzikální objekty, všechny fyzikální objekty, což jsou zbraně, interaktivní objekty a zničitelné objekty, v příslušných třídách `Weapon`, `Interactible` a `DestructibleBody`, jsou podtřídou třídy `Layers`, která vytváří abstrakci nad těmito vrstvami a definuje konkrétní vrstvy pro přátelské objekty, nepřátelské objekty, kolize, interaktivní objekty, statické objekty a prostředí.

Vrstvy pro přátelské a nepřátelské objekty jsou použity pro souboj a spolu s vrstvou statických objektů pro kolizi projektilů. Vrstva kolizí je použita pro všeobecné kolize mezi postavami, objekty a úrovní. Vrstva interaktivních objektů je použita pro detekování interaktivních objektů kolem hráče a vrstva prostředí je použita pro kolizi mrtvých částí nepřátel s úrovní.

4.4 Postavy

Postavy využívají návrhového vzoru *Component* [17, Kapitola 14] a jsou rozděleny do tří tříd `Character`, `Model` a `Controller`. Třída `Character` je zároveň podtřídou třídy `DestructibleBody`. Postavy zahrnují jak nepřátele, tak i postavu hráče. Diagram těchto tříd je na obrázku 4.13.



Obrázek 4.13: Diagram tříd postav

Třída `Controller` a její podtřídy jsou zodpovědné za ovládání postavy, která je vlastní. Tato třída a třída `EnemyController` využívají návrhového vzoru *Subclass Sandbox* [17, Kapitola 12]. Třída `Controller` svým podtřídám poskytuje funkce pro ovládání postavy a třída `EnemyController` poskytuje další funkce pro hledání cesty pomocí navigačního meshe a také se stará o zrak nepřátel. Třída `PlayerController` zpracovává uživatelský vstup.

Komunikace mezi třídou `Controller` a `Character` probíhá jednak pomocí signálů, ale také přímým přístupem k některým proměnným, jako je například směr pohybu nebo směr, kterým se postava dívá.

Třída `Model` poskytuje jednotné rozhraní pro animace postav, nezávisle na jejich konkrétní implementaci.

Tento návrh umožňuje hlavně jednoduchou změnu nepřátelských modelů a jejich umělé inteligence, ale také abstrakci některých funkcionalit, sdílených mezi nepřáteli i hráčem, jako je například pohyb nebo otáčení, do vyšších tříd.

Postavy pro klíčové události, jako je například způsobování zranění nebo vystřelení projektilů, využívají *Parameter* a *Method* stop animací.

4.5 Nepřátelé

Třída `EnemyController` kolem postav, které ji využívají, vytváří sférickou oblast, která detekuje ostatní instance třídy `Character`. Pokud do této oblasti vstoupí postava, která je ve stejné týmové vrstvě (přátelé, nepřátelé), jako postava, která ji detekuje, detekující postava k sobě připojí signál `targetAcquired` detekované postavy, který se posílá, pokud postava získala cíl, na který útočit. Tento signál se odpojí, pokud detekovaná postava z této oblasti odejde. Tímto nepřátelé „vidí“, pokud hráč zaútočí na někoho v jejich okolí.

Pokud do této oblasti vstoupí nepřátelská postava, přátelská postava, která má cíl útoku, nebo některá přátelská postava v této oblasti vyšle signál `targetAcquired`, samotná oblast je nedostačující pro další rozhodování, protože umožňuje detekovat ostatní postavy i za zdí. V těchto případech se aktivuje raycast, který je na vrstvě statických objektů, což jsou hlavně zdi, který pro každou z těchto postav v každém snímku testuje, zda je v přímé viditelnosti, do té doby, než je daná postava vidět, nebo tuto oblast opustí.

Při smrti nepřítele se jeho model rozloží na jednotlivé meshe, každému se přiřadí kolizní tvar kapsle a nastaví se jim rychlost v závislosti na množství způsobeného zranění, což vytváří efekt, že se nepřítel rozpadne. Ukázka mrtvého nepřítele je na obrázku 4.14.

Obrázek 4.14: Mrtvý nepřítel

4.6 Pohyb hráče

Pohyb postavy hráče využívá konceptu *Twin stick control*, což je způsob ovládání, kdy směr pohybu a směr útoku jsou na sobě navzájem nezávislé. Tento název vychází z herních ovladačů, kde se pro tyto dva směry ovládají levým a pravým joystickem [19].

Herní postava se otáčí a útočí ve směru kurzoru myši. Pozice kurzoru ve 3D scéně se detekuje pomocí horizontální plochy v definované výšce hráče a paprsku, vyslaného z kamery ve směru kurzoru. Bod, ve kterém tento paprsek protne tuto plochu, je pozice kurzoru.

Směr pohybu je nezávislý na směru otočení postavy a počítá se pomocí X a Y os globální transformace kamery. Těmto osám se nejprve vynuluje složka y a následně se normalizují, čímž se získají horizontální směry pohybu stejné délky.

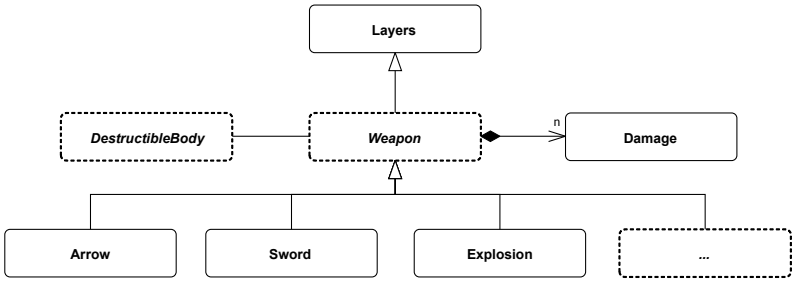
4.7 Souboj

Souboj je řešený jako interakce mezi dvěma třídami a to třídy `Weapon` a `DestructibleBody`. Třída `Weapon` zodpovídá za detekci kolize se třídou `DestructibleBody` a výpočet a způsobení poškození. Tato třída je vytvořena podle návrhového vzoru *Subclass Sandbox* [17, Kapitola 12] a svým podtřídám poskytuje funkce pro úpravu a výpočet způsobeného poškození.

Třída `DestructibleBody` řeší počet životů, přijímání poškození a delegaci informací o poškození svým podtřídám, které implementují potřebné funkce.

Poškození může být libovolně mnoho různých druhů a jsou implementovány pomocí třídy `Weapon::Damage`, která je, podobně jako třída `Weapon`, zodpovědná za výpočet poškození daného typu a poskytuje svým podtřídám funkce pro úpravu daného poškození.

Diagram těchto tříd je na obrázku 4.15.



Obrázek 4.15: Diagram tříd zbraní

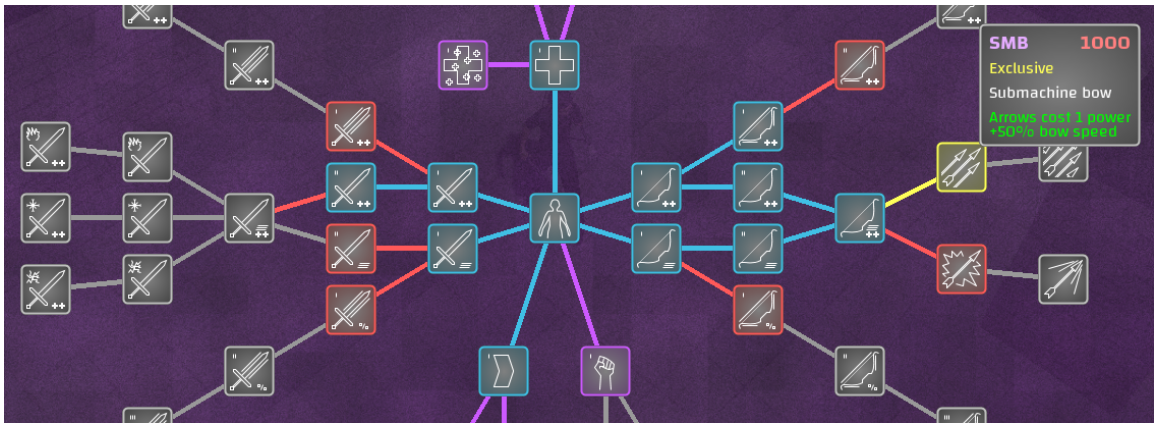
Každý typ poškození má uložené informace o základním množství poškození, násobič, šance na kritický zásah a násobič při kritickém zásahu. Každá instance třídy `DestructibleBody` má také odolnost, nebo naopak slabost proti každému typu poškození, vyjádřenou jako reálnou hodnotu, kterou se dané poškození vynásobí.

4.8 Vylepšení

Vylepšení využívají návrhového vzoru *Command* [17, Kapitola 2]. Jednotlivé efekty jsou podtřídou třídy `Upgrade::Effect` a existují čtyři druhy: `AddEffect`, `MulEffect`, `SetEffect` a `CallEffect`. První tři uvedené druhy efektů pracují s libovolným parametrem libovolného objektu a v uvedeném pořadí zvětšují, násobí nebo nastavují tento parametr. Poslední uvedený druh efektu volá libovolnou funkci libovolného objektu.

Každé vylepšení může mít libovolně mnoho efektů a také má uložený svůj název, popis, cenu a seznam textových popisů efektů.

Vylepšení jsou uložené ve stromové struktuře. Každé vylepšení je uloženo jako člen třídy `UpgradeNode`, která dále obsahuje informace o rozměrech a pozici daného vylepšení, ikoně, o rodičích a potomcích tohoto vylepšení, zda je vylepšení exkluzivní a zda je koupené.



Obrázek 4.16: Ukázka stromu vylepšení

Pokud je vylepšení exkluzivní, ze všech sourozeneckých vylepšení lze koupit pouze jedno.

Všechny instance těchto uzlů jsou uloženy v seznamu ve třídě `UpgradeTree`, která nabízí funkce pro jejich správu. Hráč k tomuto stromu přistupuje pomocí třídy `UpgradeTreeView`, která tento strom vykresluje a zpracovává uživatelský vstup.

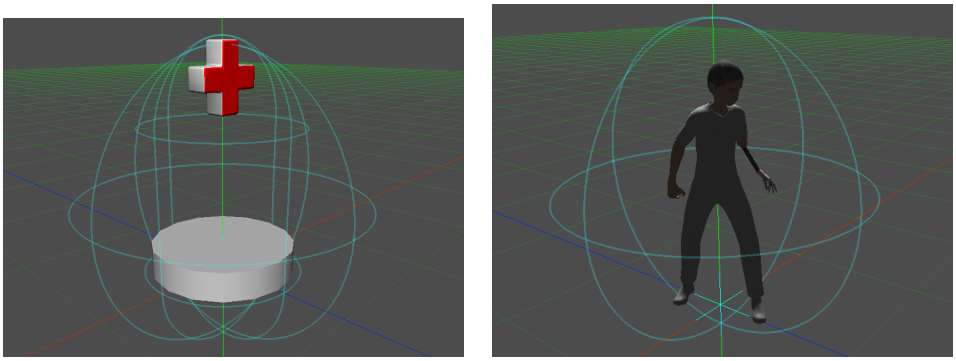
Ukázka stromu vylepšení je na obrázku 4.16.

4.9 Interaktivní objekty

Interaktivní objekty jsou podtřídou třídy `Interactible`. Každý interaktivní objekt se skládá ze tří částí: modelu, statického objektu ve vrstvě kolizí a statického objektu ve vrstvě interaktivních objektů. Každý interaktivní objekt implementuje dvě funkce. Jedna z nich definuje chování daného objektu a druhá vrací, zda se objekt dá použít.

Kolem hráče se nachází oblast, která detekuje objekty ve vrstvě interaktivních objektů a pokud je nějaký detekován a dá se použít, je na něj uložena reference a do něj je vložen efekt signalizující, že se daný objekt dá použít. Pokud hráč od interaktivního objektu odejde, nebo je objekt použit a dále se použít nedá, tento efekt je vymazán a reference je vynulována.

Ukázka interaktivního objektu včetně kolizí je na obrázku 4.17a a ukázka oblasti kolem hráče je na obrázku 4.17b.



(a) Interaktivní objekt léčení hráče

(b) Oblast okolo hráče detekující interaktivní objekty

Obrázek 4.17: Oblasti interakcí mezi hráčem a interaktivními objekty

4.10 Efekty

Efekty jsou vytvořeny podle návrhového vzoru *Subclass Sandbox* [17, Kapitola 12]. Třída `Effect` svým podtřídám poskytuje funkce pro jejich spuštění na určitém místě nebo v určitém objektu a také funkci k jejich zničení.

Efekty mohou být napsány libovolně, ale musí samy sebe odstranit.

Ukázka efektu vyléčení hráče je na obrázku 4.18.

Obrázek 4.18: Ukázka efektu

4.11 Grafické uživatelské rozhraní

Kromě stromu vylepšení hra obsahuje dva další prvky grafického uživatelského rozhraní a to minimapu a stav hráče.

Status hráče je zobrazen v levém spodním rohu a obsahuje informace o aktuálních a maximálních životech hráče, o aktuální a maximální energii a o aktuálním počtu bodů k vylepšení. Ukázka tohoto rozhraní je na obrázku 4.19a.

Minimapa se nachází v pravém horním rohu a vykresluje se každý herní cyklus. Data, která se používají pro vykreslování mapy, jsou stejná data, které používá `LevelBuilder`, přičemž ten některá data sám přidá pro využití minimapou. Ukázka této mapy je na obrázku 4.19b.

(a) Ukázka uživatelského rozhraní zobrazující stav hráče

(b) Ukázka minimapy

Obrázek 4.19: Ukázky grafického uživatelského rozhraní

Kapitola 5

Závěr

Výsledná hra implementuje všechny klíčové prvky, hlavně procedurální generování úrovní a je hratelná a snadno rozšiřitelná díky poměrně dobrému návrhu, ačkoliv, kvůli obtížnosti zadání, příliš ambicióznímu návrhu, technickým problémům a nedostatku času, hra neimplementuje některé z navržených herních prvků.

Hra obsahuje procedurálně generované úrovně, souboj mečem a lukem, několik druhů procedurálně generovaných nepřátel, dva interaktivní objekty a rozsáhlý strom vylepšení.

Jako hlavní způsob pokračování v tomto projektu vidím přechod na novější verzi Godot 3, která bohužel vyšla až v průběhu práce na tomto projektu, a která je s použitou verzí Godot 2.1 nekompatibilní, hlavně co se týče fyzikálních objektů a vykreslování. Přechod na novější verzi ovšem přinese viditelně změny v grafické kvalitě hry a výkonu kódu napsaném v jazyce GDScript.

Dále by se hra dala rozšířit v grafické stránce, přidáním chybějících textur a úpravou některých stávajících, o hudbu na pozadí a zvukové efekty, o více různých druhů interaktivních objektů a nepřátel a o možnosti změn ovládání a grafického nastavení.

Literatura

[1] Eberly, D.: Triangulation by Ear Clipping. [Online]. [cit. 2018-05-13].
URL https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf

[2] Linietsky, J.: Godot 3.0 is out and ready for the big leagues. [Online]. [cit. 2018-01-29].
URL https://godotengine.org/article/godot-3-0-released

[3] Linietsky, J.; Manzur, A.: Godot Engine - Contact. [Online]. [cit. 2018-05-02].
URL https://godotengine.org/contact

[4] Linietsky, J.; Manzur, A.: Godot Engine - Licence. [Online]. [cit. 2018-05-02].
URL https://godotengine.org/license

[5] Linietsky, J.; Manzur, A.; aj.: Custom modules in C++. [Online]. [cit. 2018-05-10].
URL http://docs.godotengine.org/en/2.1/development/cpp/custom_modules_in_cpp.html

[6] Linietsky, J.; Manzur, A.; aj.: Fixed Materials. [Online]. [cit. 2018-05-11].
URL http://docs.godotengine.org/en/2.1/learning/features/3d/fixed_materials.html

[7] Linietsky, J.; Manzur, A.; aj.: GDScript. [Online]. [cit. 2018-05-10].
URL http://docs.godotengine.org/en/2.1/learning/scripting/gdscript/gdscript_basics.html

[8] Linietsky, J.; Manzur, A.; aj.: InputEvent. [Online]. [cit. 2018-05-10].
URL http://docs.godotengine.org/en/2.1/learning/features/inputs/inputevent.html

[9] Linietsky, J.; Manzur, A.; aj.: InputEvent. [Online]. [cit. 2018-05-10].
URL http://docs.godotengine.org/en/2.1/learning/features/physics/physics_introduction.html

[10] Linietsky, J.; Manzur, A.; aj.: Introduction. [Online]. [cit. 2018-05-10].
URL http://docs.godotengine.org/en/2.1/about/introduction.html

[11] Linietsky, J.; Manzur, A.; aj.: Materials. [Online]. [cit. 2018-05-11].
URL
http://docs.godotengine.org/en/2.1/learning/features/3d/materials.html

[12] Linietsky, J.; Manzur, A.; aj.: Scenes and nodes. [Online]. [cit. 2018-05-10].
URL http://docs.godotengine.org/en/2.1/learning/step_by_step/scenes_and_nodes.html

[13] Linietsky, J.; Manzur, A.; aj.: Scripting. [Online]. [cit. 2018-05-10].
URL
http://docs.godotengine.org/en/2.1/learning/step_by_step/scripting.html

[14] Linietsky, J.; Manzur, A.; aj.: Scripting (continued). [Online]. [cit. 2018-05-10].
URL http://docs.godotengine.org/en/2.1/learning/step_by_step/scripting_continued.html

[15] Linietsky, J.; Manzur, A.; aj.: Shader Materials. [Online]. [cit. 2018-05-11].
URL http://docs.godotengine.org/en/2.1/learning/features/shading/shader_materials.html

[16] Linietsky, J.; Manzur, A.; aj.: Variant class. [Online]. [cit. 2018-05-10].
URL
http://docs.godotengine.org/en/2.1/development/cpp/variant_class.html

[17] Nystrom, R.: *Game Programming Patterns*. Genever Benning, první vydání, 2014, ISBN 0990582906.

[18] Open Source Initiative: The MIT License. [Online]. [cit. 2018-05-02].
URL https://opensource.org/licenses/MIT

[19] TV Tropes: Twin Stick Controls. [Online]. [cit. 2018-05-09].
URL
http://tvtropes.org/pmwiki/discussion.php?id=q7qwchtgv5avbl1e744i4bm0

[20] Wade, B.: BSP TREE FAQ. [Online]. [cit. 2018-05-07].
URL http://www.gamers.org/dhs/helpdocs/bsp_faq.html

Příloha A

Obsah přiloženého CD

doc/	Zdrojové kódy bakalářské práce
godot-2.1.4-custom-src/	Upravený zdrojový kód Godot enginu
licences/	Licence použitých zdrojů
src/	Zdrojové kódy hry
src/Godot32.exe	32-bitová verze Godot enginu pro spuštění hry
src/Godot64.exe	64-bitová verze Godot enginu pro spuštění hry
readme.txt	Soubor s popisem a návodem hry
video.mp4	Prezentační video
xtrbol00-hra-v-enginu-godot.pdf	Přeložená bakalářská práce

Seznam obrázků